CUTTING THRU THE HYPE:
An Analysis of Application Testing Methodologies, Their Effectiveness & The Corporate Illusion of Security

David Bonvillain
Neel Mehta
Jon Miller
Alex Wheeler
Introduction
Application Vulnerability Classes
Testing Methodologies & Solutions Analysis
  - Examples
  - Strengths
  - Challenges
  - Use Cases
Solutions
Conclusions
Most organizations are implementing application security initiatives

Wide variety of solutions and methodologies available - Many claim to ‘find all the problems’
- Application vulnerability scanning
- Static code analysis
- 3rd party penetration testing & app assessments
- Binary Analysis
- Fuzzing
- etc.

Which solutions find which issues? What are their strengths & weaknesses? What is the best methodology for different applications?

General lack of knowledge & understanding...
Reason for faults/vulnerabilities = the reason any testing solution isn’t perfect
  - (some are nowhere close)

Organizations have chosen a blind approach of “I’ll fix it if it’s a known issue or something in the LHF category of vulnerabilities”

From a software builders perspective...no company has ever gone out of business due to a security issue in their product
  - Issues can cause less sales - ISS Witty Worm
  - Issues can also increase niche business space
TRENDS IN EXOTIC CARS PURCHASED BY SECURITY EXECUTIVES
On a long enough timeline the survival rate of anything drops to zero...

Can we develop software without bugs?

- Is it worth it to develop secure software?
- Is it profitable to develop securely?
- Does secure code affect the bottom line?
- No company has gone out of business by writing insecure code

Let’s examine using our version of the Fight Club formula for applications

The number of applications in the field = A
The probable rate of failure (active exploits) = B
The average cost of business loss & developing and deploying a patch = C

\[ A \times B \times C = X \]

If \( X \) is less than the cost of the additional Q&A, coder training and 3\(^{rd}\) party security audits, it financially makes more sense to distribute insecure code.
VULNERABILITY CLASSES

Operational & Platform Vulnerabilities

Information Disclosure
OS Buffer Overflows / Missing Patches
Service Configurations
Error Handling

Implementation Vulnerabilities
- Code Injection
- Command Execution
- Information Gathering
- Error Handling

Design Vulnerabilities
- Logic Flaws
- Authorization
- Authentication
SECURITY ANALYSIS METHODOLOGY
LEVEL OF AUTOMATION

Manual vs. Automated
HAL: “Let me put it this way, Mr. Amor. The 9000 series is the most reliable computer ever made. No 9000 computer has ever made a mistake or distorted information. We are all, by any practical definition of the words, foolproof and incapable of error.”

2001: A Space Odyssey
SECURITY ANALYSIS METHODOLOGY
TARGET TESTING STATE

Static (Off-Line) vs Dynamic (Runtime)
Automated Dynamic
- e.g., Fuzz Testing, Vulnerability Scanning

Automated Static
- e.g., Source/Binary Code Scanning

Manual Dynamic
- e.g., Parameter Tampering and Social Engineering

Manual Static
- e.g., Source/Binary Code Auditing
AUTOMATED DYNAMIC TESTING
Programmatic Analysis of a Runtime Target for Security Issues

Common Components:
- Trigger: inputs to invoke security issue conditions
- Indicator: anomaly evidencing security issue
- Runtime Engine: controls the firing of triggers and observing of indicators
- **Fuzz Testing** - Noting defects by observing failures generated by programmatically submitting arbitrary data to program inputs.

- **Vulnerability Scanning** - Programmatically submitting transactions from a data set of inputs and outputs mapped to known issues.

- **Application Scanning** - A combination of both approaches, where inputs are fuzzed with data for known classes of issues.
AUTOMATED DYNAMIC SECURITY TESTING
SCALE OF COMPLEXITY FOR EXAMPLES

Fuzz Testing
Application Scanning
Vulnerability Scanning
False Positives
- Runtime provides inherent benefits
  - Interpretation can still be an issue

Reliability & Consistency
- Programmatic approach ensures reliable and consistent application of tests (including mistakes), useful in developing baselines

Resource Requirements
- Scanning vs. Fuzz Testing
Weak Assurance (Positive & Negative)
- No Fault != No Flaw
- Unknown level of unexercised code data permutations

Unknown Level Coverage
- Only code audit can provide a baseline for measurement

Low Flexibility
- Unexpected circumstances cannot be addressed without additional programming
Fuzz Testing

- Pre-production
- Sparsely audited code base
- Complex application input processing
- Weak, immature, or informal SDLC
- Large amount of observable indicators
- Prior runs yield numerous significant results
Application Scanning
- Strongly typed flaw classes
- Deterministic & observable behavior
- Generally known input types
- Prior runs yield numerous significant results

Vulnerability Scanning
- Deterministic & observable behavior
- Known transaction sequences
- Strong trigger to indicator mappings
Fuzz Testing

- Mature & widely deployed code base
- Low fault observation accuracy or ability
- Thoroughly audited code base
- Prior runs yield no significant results
- Largely unknown program inputs
MS07-010
- Default Enabled in Vista
- Integer Overflow in Protection Engine Library PDF Parser affecting multiple products
- Simple Issue with complex data flow
- Discovered in Static Binary Analysis
  - Fuzz Testing would have needed multiple encoding support
  - Source Testing would have needed
Application Scanning
- Substantial variability around program inputs
- Low visibility into issue indicators
- Built with non-standard/custom technology

Vulnerability Scanning
- Highly customized services environment
- Low confidence in response accuracy
AUTOMATED STATIC TESTING
An automatic static analysis tool discovers security issues in code (src/binary), when run with minimal or no user interaction.

Numerous commercial tools, open source tools, academic papers and work in the field of automated static analysis.

Administrations run a quick static analysis of their application at an appropriate point in the development lifecycle, and then respond to the results.
Evaluation procedure:
- Select a legacy version of an application (closed-src), containing known but private vulnerabilities.
- Evaluate the coverage of the tool over known issues.

Less fair evaluation procedure:
- Select a current version of a widely-deployed and scrutinized application with privately known 0day issues (Apache, Firefox 3.08, etc.)
- Evaluate their competence, relative to the state of the art attacks these applications constantly face.
Informal flaw identification:
- Antiquated pattern-matching solutions (context-away or grep).

Formal verification methods:
- Model-checking solutions.
- Data-flow analysis solutions.
- Abstract interpretation-derived solutions.
AUTOMATED DYNAMIC SECURITY TESTING
SCALE OF COMPLEXITY FOR EXAMPLES

- Involved
- Simple
- Abstract Interp.
- Data Flow Analysis
- Model Checking
- Pattern Matching
Locating low-context flaws:

```php
my_table = req->getParameter("unfiltered");
$db->query("SELECT * FROM " , my_table,"WHERE intent = "EXPOSE ALL MY DATA");
```

- Quite useful if you left assessing enormous volumes of terrible code.

**Speed, human interaction:**

- Fast, little to no human interaction during scans

**Integrates well with most development life-cycles.**
Tool-specific challenges:
- applications without source code, binaries without information to return to source
- no application support for your language
- SAT that are not tightly integrated with the build processes are at a disadvantage
- SAT applications that perform 'pseudo-compilation' are dangerously deficient and vulnerable to asymmetries

High noise ratios:
- Balancing false positives and negatives
- An application that discovers 1 single serious security issue, and 10,000 non-issues is useful?
- Tuning may help, we wish you luck.
Two extremely high level problems, neither simple for automated SAT:

1) Developing and correctly expressing a set of security-critical invariants, which if disproven are issues.
   - It’s challenging to express high-level criteria or requirements as program invariants.
   - It is rarely easy to define all critical invariants for any sufficiently large application manually, let alone via automatic SAT.
   - Invariants are typically a large relatively static vendor-provided list, woefully limited to issues they can confidently detect.
2) Developing an interpretation of the application that lends itself to proving or disproving invariants.

- Abstract interpretation is largely a purpose-driven approach, tailored to the invariants you’re looking to prove/disprove.
- Abstract interpretation to prove a single invariant might be simple, but is quickly complicated by interprocedural analysis, undecidable data structures or storage mechanisms.
- Model checking is limited to a crippling subset of operations in any modern application.
STRONG USE CASES

- Timely, and sometimes resource-efficient detection of blatantly-simple flaws in enormous code bases.
- As part of a dev lifecycle, quickly detecting regression or re-introduction of blatantly-simple flaws.
- For applications where the risk profile is limited to none, that do not warrant alternate forms of testing.
Obtaining strong assurance about the security of a critical application in the face of a skilled and motivated attacker.

Against a code base that has undergone any degree of more sophisticated review.

In the hands of a developer who cannot interpret or filter reports correctly.

- Such as when deciding to remove code with memory leaks from PRNG’s.
MANUAL DYNAMIC TESTING
Human-navigated application usage. Generally focused on one of the following:
- Manual fuzz-testing - discovering unanticipated implementation flaws.
- Assurance validation.
- Verifying implementation against specification.

Almost always aided by test tools.

Test cases come almost exclusively from the tester.

Critical background information provided by developers.
GENERAL STRENGTHS

- Draws on the intuition of the tester (capacity for parallelism in thought).
- Much of manual security testing is pattern recognition, an inherently subconscious process.
  - Innocuous, seemingly irrelevant inconsistencies often reveal large and severe underlying flaws.
- Tests live implementations, so false positives are reduced.
- Directly emulates the process of a malicious attack performed without source.
Can be time consuming for large and complex applications.

- Application risk profile, relative to size of critical attack surface and complexity, must be favorable to justify in-depth testing.

Might include a steep learning curve.

Heavily dependent on the tester:

- How orthogonal their security testing skillset and methodology is to the application’s vulnerability set.

Testing environment may not mirror production.
A highly experienced security researcher or consultant, properly scoped:
- High risk applications, or high-risk portions of the attack surface for larger applications.

Especially critical to use manual dynamic testing in cases where:
- Attackers are expected to be blindly attacking a high-risk application.
- Results of test cases that fail cannot be easily identified through automated testing.
- An application that is inherently risky will almost always require this form of testing (especially new and untested technologies).
Applications with limited or no feedback, or asynchronous feedback

The wrong tester, or the wrong application for the tester

Cases where the requirements of an assessment doesn’t match the expected risk profile for an application
SSH CRC32 Compensation Attack (CVE-2001-1044)
- Discovered by Michal Zalewski:

From Bugtraq post Feb. 2001:

\$ ssh -v -l `perl -e '{print "A"x88000}'` localhost

Remote, pre-authentication, default remote vulnerability in SSH.COM and OpenSSH daemons, at the peak of their usage.

Actual issue:
- 16-bit integer truncation deep in code designed to correct a less serious protocol weakness.
- Extremely subtle for the time, and unlikely to be found by other methods.
MANUAL STATIC TESTING
Human Review of a Non-Running Target for Security Issues

Common Components:
- Target documentation (architecture, implementation, configuration)
- Offline toolset (code browser, disassembler, graphing tools)
Source Code Audit

```
In any case, the prev > end check must be:
if (code != end + 1 || prev > end) {
    strm->msg = (char *)"invalid Izw code";
    return Z_DATA_ERROR;
}
match[stack++] = (unsigned char)final;
code = prev;
}
```

Configuration Audit

```
lp@:4:7:lp:/var/spool/lpd:
sync@:5:0:sync:/sbin/bin/sync:
shutdown@:6:0:shutdown:/sbin/shutdown:
halt@:7:0:halt:/sbin/sbin/halt:
mail@:8:12:mail:/var/spool/mail:
news@:9:13:news:/var/spool/news:
```
MANUAL STATIC SECURITY ANALYSIS
GENERAL STRENGTHS

Strong Assurance Potential
- Known data and code points allow baseline

High Coverage Potential
- Without resource considerations

Flexibility
- Adaptable skill & tool set
MANUAL STATIC SECURITY ANALYSIS
GENERAL CHALLENGES

Accuracy Issues
- False positives: without verification step, many issues cannot be triggered
- Missing: humans make mistakes

High Resource Requirements
- Skill-based methodology, with high demand

High Error Factor
- Same factors introducing flaws are also at work here

Inconsistency
- Same auditor may miss or hit the same flaw on different days.
Manual Code Audit

- Access to overlapping skilled resources for repeat engagements
- Prior automated tests returned only minor findings
- Largely non-standard/custom program inputs
Configuration Review

- Low risk of setting values changing in runtime (e.g., malware or backdoor)
- Largely known data sources and formatings
- Availability of job aids for reduction of effort (e.g., grep, work plans, or checklists)
struct igmp_report
{
    __u8 type;
    __u8 resv1;
    __be16 csum;
    __be16 resv2;
    __be16 ngrec;
    struct igmpv3_grec grec[0];
};
Generate_Report ( ... )
  igmp_report *report = arg_0;
  SLIST *addrlist = arg_4;
  unsigned short cnt;

  for(addrlist = addrlist->nxt, cnt=0; report->nxt; cnt++);

  report = malloc(cnt*sizeof(report->ngrec) + sizeof(*report));
  for(addrlist = addrlist->nxt, cnt=0; report->nxt; cnt++)
    memcpy(report->ngrec+cnt, addrlist, 4)
WASC Statistics Project: Consolidated analysis of common vulnerabilities across a variety of web applications

- Statistics based on over 32,000 sites and 70,000 vulnerabilities of different degrees of severity

- 2 different data sources:
  - Automated vulnerability scanning testing results
  - Combination / Grey-Box Testing methodology:
    - Application vulnerability scanning coupled with manual analysis, manual search for vulnerabilities which cannot be detected by automated scanner, and source code analysis.

- 3 data sets were obtained:
  - Overall statistics
  - Automated scanning statistics
  - Black and White-Box methods security assessment statistics
    - Grey-Box testing was limited to interactive web applications

(http://www.webappsec.org/projects/statistics/)
Results:

- Probability to detect high risk vulnerabilities using combined testing methodologies is 12.5 times higher than using automated scanning.
- Over 7% of analyzed sites can be compromised automatically.
- Using combined/grey-box methodologies high severity probability reaches 96.85%.
Recent Consulting Project Dataset
- 2 Representative Applications used - PHP and J2EE

Application testing methodologies analyzed across multiple vendor types
- MSSP
- Static Code Analysis Tools
- Automated Dynamic Scanning
- Consulting Vendors

Present vulnerabilities analyzed and then additional implanted across all vulnerability classes and ranges of severity
Chart of solutions overall ability to identify vulnerabilities when compared as a whole

- MSSP
- SAT
- Dynamic
- Automated
- Combination / Consulting

PHP WebApp Weighted Score
J2EE WebApp Weighted Score
Solutions overall ability to find vulnerabilities within particular vulnerability class
Chart for solutions ability to find high severity vulnerabilities across all classes

- **Data Validation**
- **Authorization**
- **Injection Attacks**
- **File Handling**
- **Encryption Handling**
- **Session Management**
- **Exception Management**

**Legend:**
- MSSP
- SAT
- Dynamic Automated
- Combination
You must determine risk to establish testing methodology.

Spending more on security than the overall liability is a waste of time, resources and money.
PUBLIC RISK FORMULAS

- **Risk** = 
  - Threat x Vulnerability x Impact 
  - Asset Value x Threat 
  - Confidentiality x Integrity x Availability x (Threat x Vulnerability) 
  - Probability x Damage Potential (Microsoft)

- Seriously? 
- How are these ideas defined? 
- How do I rank CIA? 
- Great idea, stupid implementation 

E=MC Retarded
The only risk that matters is financial...

Intro...
- Microsoft
- Understand (Asset / Threat / Vulnerability / Attack / Countermeasure)
- DREAD Ranking
  - Damage Potential
  - Reproducibility (only needs to happen once)
  - Exploitability
  - Affected Users
  - Discoverability
- What about money??
  - That’s all I care about...
Threat modeling is determining risk

- Business criticality / risk modeling
  - Exposure to attack
  - Business criticality
    - Effect to business
    - Effect to customers / reputation
    - Effect to personal information / exposure
    - Financial loss impact

1. Identify Security Objectives
2. Application Overview
3. Decomposed Application
4. Identify Threats
5. Identify Vulnerabilities
WHAT SOLUTION DO WE USE?

- Automated / Static / Dynamic / Manual

Questions to ask:

1. Maturity of your program
2. Skill level of personnel
3. Availability of skilled hours
4. Maturity of the application
5. Availability of code
6. Complexity of the application
7. Technology / language
8. Availability of test resources
9. Volume of users
10. Internal vs. external facing
11. Data sensitivity
12. Sensitive functionality
13. Regulatory requirements
WHAT SOLUTION DO WE USE?

- Sweet... we answered those questions... now what?
- Use common sense, there is no magic formula.. (at least we haven’t been able to figure out something perfect)
Coming Soon: Form based calculator...

Based on the ‘MSAMACTA’ formula outlined earlier, input variables on your application, and it will recommend the best testing methodology.
<table>
<thead>
<tr>
<th>Testing Solution</th>
<th>Strengths</th>
<th>Weaknesses</th>
<th>Process Integration</th>
</tr>
</thead>
</table>
| Automated Testing - Dynamic Environment (Vulnerability Scanning) | • Quickly identifies implementation vulns  
• Can identify Operational and Platform vulns | • Many false positives  
• Most design vulns missed  
• Noisy traffic for IDS systems  
• Can impact resources | During testing phase or within post-production deployment environment |
| Automated / Manual - Dynamic Environment (Penetration Testing) | • Tests actual implementation  
• Finds issues from an attackers perspective  
• Can find Implementation, Design and Operational vulns | • Can be slow  
• Difficulty with some implementation vulnerabilities  
• Testing can impact production | During testing phase or within post-production deployment environment |
| Threat Modeling                                      | • Quickly identifies Design vulnerabilities  
• Can be implemented early in dev cycle | • Ineffective for Implementation and Operational vulns  
• High personnel impact | Requirements analysis and security design phases of the SDLC |
<table>
<thead>
<tr>
<th>Testing Solution</th>
<th>Strengths</th>
<th>Weaknesses</th>
<th>Process Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Testing - Static Environment</td>
<td>• Detailed remediation info</td>
<td>• Comprehensive approach can be time consuming</td>
<td>During the coding phases of the SDLC or as a component of a comprehensive blended assessment</td>
</tr>
<tr>
<td>(Manual Code Review)</td>
<td>• Some methods can quickly identify LHF issues</td>
<td>• Can require high personnel involvement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Able to provide deeper analysis to show impact</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automated Analysis - Static Environment</td>
<td>• Quickly identifies pattern match vulnerabilities</td>
<td>• Few actionable results</td>
<td></td>
</tr>
<tr>
<td>(Static Source Code Review Tools)</td>
<td>• Often faster and cheaper than a manual review</td>
<td>• Cannot find Design vulns</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Cannot find certain classes of Implementation vulnerabilities</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehensive Blended Assessment</td>
<td>• Efficiency</td>
<td>• Cost and duration</td>
<td>QA &amp; Post Production</td>
</tr>
<tr>
<td>Methodology</td>
<td>• Accuracy</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
There is no real ‘solution’
No single ‘solution’ comprehensively identifies all critical application vulnerabilities or across all vulnerability classes.
A comprehensive program should include a blend of all of the various testing methodologies available.
Apply the appropriate testing methodology based on factors such as:
- Application Risk Profile
- Criticality
- Timeframe
- Availability of Resources
- Budget